Conditional Blockchain Rewriting by Message-Controlled Chameleon Hash

Yangguang Tian
Osaka University
sunshine.tian86@gmail.com

Tsz Hon Yuen
University of Hong Kong
johnyuenhk@gmail.com

Yingjiu Li
University of Oregon
yingjiul@uoregon.edu

Binanda Sengupta
A STAR*
binujucse3@gmail.com

***Abstract*—Rewriting a blockchain is useful when a transaction in a block contains illegal or harmful hashed messages that should be modified or deleted by a permitted party, while the consistency of the subsequent blocks in the blockchain is maintained. However, selecting arbitrary messages for rewriting is not desirable as it may defeat the purpose of rewriting.**

In this work, we introduce the notion of *message-controlled chameleon hash* (MCH) to control the message to be updated in the blockchain. We propose an MCH construction which is secure against arbitrary or malicious rewriting on the hashed messages for blockchain rewriting. Our contributions are three-fold. First, the proposed MCH enables a permitted party to select a candidate message from a specified message set for blockchain rewriting at message level. Second, the proposed MCH allows the amount of a transaction to be modified without causing transaction inconsistency. Third, the proposed MCH can be easily integrated into both permissioned and permissionless blockchains.

***Index Terms*—Blockchain Rewriting, Chameleon Hash, Message Control**

I. Introduction

Blockchain, a distributed ledger technology, has received tremendous attention from research communities as well as from industries in recent years. Blockchain was first introduced in the context of Bitcoin [21], where all payment transactions are appended in a public ledger (i.e., the Bitcoin blockchain), and each transaction is verified by network nodes in a peer-to-peer manner. Blockchain grows by one block at a time, and is an append-only structure of blocks with a hash-chain, where the hash of a block is linked to the next block in the chain. Each block consists of a set of transactions which is mapped to a single hash value for the block using the Merkle tree [20]. Each transaction, on the other hand, contains various objects which need to be recorded in the blockchain.

One of the notable properties of a blockchain is "immutability", i.e., the objects already recorded in the blockchain cannot be modified. However, blockchain rewriting is necessary in many scenarios, sometimes due to certain legal obligation such as the General Data Protection Regulation (GDPR) introduced recently [4]. An application scenario is that the payment-transactions of customers in a business are required to be recorded in a blockchain, where certain recorded transactions may later need to be modified under certain conditions. One typical example is that customers may receive cashback on shopping (e.g., for timely payment of taxes) after making payments. Another typical example is that certain advance payments need to be adjusted later due to inevitable changes of exchange rates or business conditions. For these examples, the transactions cannot be modified arbitrarily but under certain conditions that restrict the modified transaction amount.

Motivation. There are two research questions regarding blockchain rewriting: 1) which party can rewrite (i.e., access control); 2) which object in a transaction is allowed to be rewritten and how to rewrite it (i.e., object control). In this work, we focus on the second research question about object control in the permissioned and permissionless blockchains.

Object control is essential to blockchain rewriting. Let us consider a *buy limit order* transaction in the market, where a payer can buy a cryptocurrency at a specified price or better. A payer is willing to buy one BTC with ETH with the exchange rate from 1:30 up to 1:32. The payer creates a transaction containing a statement "I spend ETH 30 to buy BTC 1 from the payee" and appends it to the blockchain. The payer additionally selects an *object set* including monetary values ranging from ETH 30 to ETH 32, and links the object set to that transaction. Later, the payee chooses a value from the object set to rewrite the transaction amount. The payee may change the amount from ETH 30 to ETH 31 if the market rate is 1:31 when the exchange takes place.

Arbitrary or malicious rewriting by the permitted party (who has the rewriting privilege) is dangerous. For example, a permitted party may abuse her rewriting privilege and replace the recorded objects with incorrect ones, such as "I spend ETH 100 to buy BTC 1 from the payee". In a permissionless blockchain, the payer (or the transaction owner) is supposed to modify his own created transaction. Specifically, the payer creates/modifies a transaction amount according to an object set, which is selected by the payee. In this work, we allow any portion of the statement (we call it message for convenience) contained in a transaction to be modified in a controlled manner. Specifically, we show how to modify monetary values inside a statement.

To realize the message control for blockchain rewriting, we re-design the underlying cryptographic primitive: chameleon hash [19]. Chameleon hash enables a permitted party to rewrite the hashed message without changing the hash output and without breaking the links between consecutive blocks in a chain. In this work, the core objective is to design a message-controlled chameleon hash (MCH), which ensures that: 1) a

message m can be rewritten if the newly selected message m' is selected from a designated message set $\mathsf{M} = \{m, m'\}$ specified by an honest party such that

$$\mathsf{Hash}_{\mathtt{pk}}(m; r) \quad = \quad \mathsf{Hash}_{\mathtt{pk}}(m'; r'),$$

where pk is the chameleon public key (sk is the corresponding trapdoor), and r, r' are randomness; and 2) the proposed MCH can be integrated into permissioned and permissionless blockchains.

Technical Challenges. The first challenge is to prevent the permitted party from finding collisions with respect to an arbitrarily selected message. Since the permitted party has the rewriting privilege (i.e., holding the trapdoor sk), she can always find a collision for any arbitrary message which may appear in the blockchain. This contradicts our design goal. The ideal solution is to enforce a control on the message selection at the time of collision finding.

The second challenge is to address transaction inconsistency if blockchain rewriting occurs on the transaction amount. The rewritten message inside a transaction is varied, including outscript, transaction amount, and timestamp [2]. If the rewritten message modifies the transaction amount, it may lead to transaction inconsistency [14], [23] (i.e., changing spendable data of a transaction affects other transactions in the chain). To the best of our knowledge, transaction inconsistency has never been rigorously addressed in the context of blockchain rewriting.

Our Techniques. We now explain our key technical insights to address those challenges. First, we apply a message-controlled technique to the collision finding algorithm. Specifically, MCH requires two trapdoors to find a collision. One is the chameleon trapdoor sk; the other is a *relationship value*, which is derived from an original message and every new message. The original message is the hashed one present in the chain that is to be replaced by one of the new messages. We call them in general candidate messages, which belong to the message set. The relationship value between the original and new messages can be regarded as the second trapdoor, which serves as an input to the collision finding algorithm.

We rely on a chameleon hash-based accountable assertion to generate relationship values between candidate messages. Accountable assertion means that an assertion is held accountable for a message. It was initially used to prevent double-spending attacks: the corresponding trapdoor could be easily extracted if two pairs of collided (message, assertion) values are given [24]. We use the extracted trapdoor as a relationship value. In particular, the relationship between these two messages is publicly verifiable, which enforces the trapdoor holder to select a message from M to find a valid collision.

Second, we use the time-lock mechanism [1] to address transaction inconsistency when the action of rewriting affects transaction amounts. The key idea is to allow rewriting of a transaction's amount prior to a specified timestamp by which the transaction is locked. Once the timestamp is reached, the transaction becomes *spendable* and it cannot be rewritten further.

We integrate the proposed MCH into mutable blockchains using the existing consensus mechanisms (e.g., PoW). Specifically, we slightly modify the transaction structure by including a bit $b \in \{0, 1\}$ to distinguish between normal transactions and revision-transactions, and the network nodes synchronize the transactions at their end according to the bit b. When a transaction appears in a valid new block appended to the blockchain (e.g., through the PoW consensus), each node in the network regards it as a normal transaction if its bit b is zero; otherwise, each node treats it as a revision-transaction (or dummy transaction) specifying the changes pertaining to a previous transaction, and modifies that transaction accordingly (if its time-lock has not expired) in the local copy of the chain stored at its end.

Our Work. The major contributions are summarized as follows.

- *Generic Framework.* We introduce the first generic framework of message-controlled chameleon hash (MCH), which is used to secure message-level blockchain rewriting. The control on the rewritten message is essential to blockchain rewriting because it prevents a chameleon trapdoor holder from modifying the hashed message maliciously.
- *Practical Instantiation.* We present a practical instantiation of MCH, and we show the detailed evaluation to validate its practicality. The evaluation shows that MCH incurs a moderate computational and storage cost as compared to the standard chameleon hash.
- *Applications for Blockchain Rewriting.* We show that MCH can be integrated into both permissioned and permissionless blockchains for message-level rewriting. The proof-of-concept implementation shows that blockchain rewriting based on our approach incurs almost no overhead to chain validation when compared to the immutable blockchain.

A. Related Work

Blockchain Rewriting. Blockchain rewriting was firstly introduced by Ateniese et al. [10]. The proposed solution enables a block in a blockchain to be modified by a permitted party, by replacing the use of regular SHA256 hash function with a chameleon hash (CH) in block generation [19]. The hashing of CH is parametrized by a public key pk, and CH behaves like a collision resistant hash function if the secret key sk (or trapdoor) is unknown. A permitted party who has the trapdoor can find collisions and output a new (message, randomness) values without changing the hash value.

Camenisch et al. [11] introduced a new cryptographic primitive: chameleon hash with ephemeral trapdoor (CHET), which can be regarded as a variant of CH. CHET requires that a permitted party must have two trapdoors for rewriting: one trapdoor sk is associated with the public key pk; the other one is an ephemeral trapdoor chosen by the party who initially computed the hash value. CHET provides more control in rewriting in the sense that the party, who computed the hash value, can decide whether the holder of sk shall be able to rewrite the hash by providing/withholding the second trapdoor.

Derler et al. [13] proposed a policy-based chameleon hash (PCH) that supports fine-grained blockchain rewriting. PCH replaces the public-key encryption scheme in CHET by ciphertext-policy attribute-based encryption (ABE) scheme. A party possessing a chameleon trapdoor can modify the transaction if hers attribute set corresponding to the chameleon trapdoor satisfies the embedded access policy. PCH enables rewriting at a transaction-level – which is more practical compared to block-level rewriting as described in [10]. However, PCH is applicable to permissioned blockchain only, since a key generation center (KGC) is needed for ABE. Recently, Tian et al. proposed an accountable PCH for blockchain rewriting (PCHBA) [29]. The proposed PCHBA enables the modifiers of transactions to be held accountable for the modified transactions. In particular, PCHBA allows a third party (e.g., KGC) to resolve any dispute over modified transactions.

In another work, Puddu et al. [23] proposed μchain: a mutable blockchain. A transaction owner introduces a set of transactions, including an active transaction and multiple inactive transactions, where the inactive transactions are possible versions of the transaction data (namely, mutations) encrypted by the transaction owner. The decryption keys are distributed among miners using Shamir's secret sharing scheme [27]. The transaction owner enforces access control policies to define who is allowed to trigger mutations in which context. Upon receiving a mutation-trigger request, a set of miners runs a Multi-Party Computation (MPC) protocol to reconstruct the decryption key, decrypt the appropriate version of the transaction and publish it as an active transaction. μchain incurs considerable overhead due to the use of MPC protocols across multiple miners.

Deuber et al. [14] introduced an efficient redactable blockchain in the permissionless setting. The proposed protocol relies on a consensus-based e-voting system [18], such that the modification is executed in the chain if a modification request from a user gathers enough votes from miners (we call it V-CH for convenience). When integrated into the existing Bitcoin system, V-CH incurs only a small additional overhead. In a follow-up work, Thyagarajan et al. [28] introduced a protocol to repair blockchains (e.g., fix buggy contracts, or remove illicit contents), which acts as a publicly verifiable layer on top of any permissionless blockchain. The proposed protocol allows a user to propose a repair first. Then, a group of users output a decision on the repair proposal after a publicly verifiable deliberation process. Eventually, miners check whether the decision adheres to the repair policy and update the contents in the repaired block.

In this work, we use chameleon hash cryptographic primitive to secure blockchain rewriting. Table I shows that our proposed MCH supports a controlled rewriting at the message-level in both permissioned and permissionless settings. Message-level rewriting means that an authorized modifier can modify various messages (e.g., transaction amount) inside a mutable transaction, which is more fine-grained than block-level and transaction-level modifications. Message-controlled rewriting means that a message in a transaction can be modified in a controlled manner (to be precise, the modification value must be chosen from a message set only). We stress that if a special symbol $\perp$ is included in every message set, the modifier may remove the illicit content by replacing the original message by $\perp$.

TABLE I: The comparison between the blockchain rewriting solutions. CH-based means CH-based blockchain rewriting. Blockchain rewriting can be performed in a permissioned/permissionless (perm/perm-less) setting, or both of them.

	CH [10]	μchain [23]	PCH [13]	V-CH [14]	PCHBA [29]	Ours
CH-based	✓	×	✓	×	✓	✓
Setting	both	both	perm	perm-less	perm	both
Granularity	block	trans	trans	trans	trans	msg

II. Building Blocks

In this section, we introduce chameleon hash functions and accountable assertions, which are used in our proposed construction.

A. Chameleon Hash

We will use chameleon hash function for blockchain rewriting. A chameleon hash (CH) function is a randomized hash function that is collision resistant but provides a trapdoor [19]. Given the trapdoor $\mathtt{sk}$, a message m with a randomness r, and a new message m', it computes a new randomness r', such that $h = \mathsf{Hash}_{\mathtt{pk}}(m; r) = \mathsf{Hash}_{\mathtt{pk}}(m'; r')$. We call them collided (message, randomness) values (or collision).

Definition 1: A public-coin chameleon hash (CH) function consists of the following algorithms.

- $\mathsf{KeyGen}(1^\lambda)$: It takes a security parameter λ as input, outputs a chameleon key pair $(\mathtt{sk}, \mathtt{pk})$.
- $\mathsf{Hash}(\mathtt{pk}, m)$: It takes the chameleon public key $\mathtt{pk}$, and a message $m \in \mathcal{M}$ as input, outputs a chameleon hash h, a randomness r. Note that $\mathcal{M} = \{0,1\}^*$ denotes a general message space. We sometimes write $h = \mathsf{Hash}_{\mathtt{pk}}(m; r)$ when we talk about collision.
- $\mathsf{Verify}(\mathtt{pk}, m, h, r)$: It takes the chameleon public key $\mathtt{pk}$, message m, chameleon hash h and randomness r as input, output a bit $b \in \{0,1\}$.
- $\mathsf{Adapt}(\mathtt{sk}, m, m', h, r)$: It takes the chameleon secret key $\mathtt{sk}$, messages m, m', chameleon hash h and randomness r as input, outputs a new randomness r'.

We assume that the Adapt algorithm always verifies if the chameleon hash h given is valid, or it outputs $\perp$. Correctness is straightforward and is given in [19].

For security, chameleon hash function needs to achieve collision-resistance and uniformity. Informally speaking, collision-resistance means that any malicious party cannot find two pairs (m, r) and (m', r') that map to the same chameleon hash h. Uniformity requires that the output of Hash is uniformly distributed. For a uniformly random value r, the new randomness $r' = \mathsf{Adapt}(\mathtt{sk}, m, m', h, r)$ is also a uniformly distributed random value. The formal definition is given in [19].

A secret-coin CH is different from a public-coin CH in that a check string ξ is output from the function Hash. The check string ξ replaces the randomness r in Verify and Adapt [10].

1) Semi-public-coin Chameleon Hash: In this paper, we use a new variant called *semi-public-coin* CH with $(h, \xi) \leftarrow \mathsf{Hash}(\mathtt{pk}, m)$, and use the check string ξ to replace the randomness r in Verify only. The output of Adapt is modified to $(r', \xi') \leftarrow \mathsf{Adapt}(\mathtt{sk}, m, m', h, r)$.

Consider the discrete logarithm (DL)-based CH with public key $\mathtt{pk} = (g, \chi = g^{\mathtt{sk}})$. Then $\mathsf{Hash}(\mathtt{pk}, m)$ outputs $h = \chi^m g^r$ and $\xi = (R = g^r, \pi)$, where π is a NIZK for the value $\log_g R$. In Verify, the hash value h can be verified by checking if $h = \chi^m R$ and if π is valid. In Adapt, we compute $r' = (m - m') \cdot \mathtt{sk} + r$ and $\xi' = (R' = g^{r'}, \pi')$, where π' is a NIZK for $\log_g R'$. Hence, it is a semi-public-coin CH.

B. Accountable Assertions

The accountable assertion (AA) was proposed in [24] to bind messages in an accountable way. If the attacker asserts two contradicting messages, then an assertion secret key ask can be extracted.

Accountable assertion includes two security guarantees: extractability and secrecy. Extractability guarantees that the trapdoor is extracted given two pairs of (message, assertion) values. Secrecy prevents the extraction of the trapdoor if no collision is given.

The concrete construction of AA in [24] is based on the idea of chameleon authentication trees (CAT) [26]. Loosely speaking, it is a combination of collision resistant hash functions H and chameleon hash functions CH to build a Merkle-style tree. Let chameleon hash be $(h, r) = \mathsf{Hash}(\mathtt{apk}, m)$ for some chameleon public key apk. We show a simple tree with 3-arity in Figure 1. Specifically, each tree node has three slots (A, B, C) and each slot may have a child (represented by a solid line). Initially, each leaf slot is associated with a dummy (message, randomness) values $(u_{i,j}, v_{i,j})$, where $i \in \{1, 2, 3\}$ is the depth of the node, and $j \in \{1, \cdots, 6\}$ is the position of the node. E.g., $(\overline{B_{3,4}}, v_{3,4}) = \mathsf{Hash}(\mathtt{apk}, u_{3,4})$.

We show the assertion operation in Figure 1. Assume we assert a message m to a slot $B_{3,4}$. To do so, we compute the first collision in $(B_{3,4}, v_{3,4}) = \mathsf{Hash}(\mathtt{apk}, u_{3,4})$. That is, we use the trapdoor ask to find a randomness $r_{3,4}$ such that $B_{3,4} = \mathsf{Hash}_{\mathtt{apk}}(u_{3,4}; v_{3,4}) = \mathsf{Hash}_{\mathtt{apk}}(m; r_{3,4})$. Next, we need to compute the elements $A_{3,4}, C_{3,4}, A_{2,2}, B_{2,2}, C_{2,2}, A_{1,1}, B_{1,1}C_{1,1}$ which are the slots along the assertion path from $B_{3,4}$ to the root. These elements can be computed from the corresponding values $(u_{i,j}, v_{i,j})$. To assert $(A_{3,4}, B_{3,4}, C_{3,4})$ to the parent $A_{2,2}$, we compute $h_{3,4} = \mathtt{H}(A_{3,4}, B_{3,4}, C_{3,4})$. Again, we use the trapdoor ask to find a randomness $r_{2,2}$ such that $A_{2,2} = \mathsf{Hash}_{\mathtt{apk}}(u_{2,2}; v_{2,2}) = \mathsf{Hash}_{\mathtt{apk}}(h_{3,4}; r_{2,2})$. We repeat this procedure up to the root. In this example, the assertion τ on message m includes three randomness $(r_{3,4}, r_{2,2}, r_{1,1})$ so that any public user can verify the (message, randomness) values along the assertion path from $B_{3,4}$ to the root. This completes the informal description of the underlying assertion data structure.

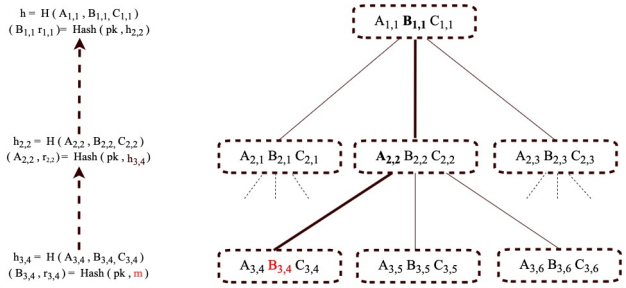

Fig. 1: An assertion in a Merkle-style tree. H indicates collision resistant hash function, Hash means chameleon hash function CH.

If the trapdoor holder asserts two messages in slot $B_{3,4}$, anyone can extract ask because there exist two pairs of collided (message, randomness) values such that $B_{3,4} = \mathsf{Hash}_{\mathtt{apk}}(m; r_{3,4}) = \mathsf{Hash}_{\mathtt{apk}}(m'; r'_{3,4})$, where $r'_{3,4}$ is a randomness associated with message m'. Similarly, if two pairs of collided (message, randomness) values occur in slot $A_{2,2}$, the trapdoor can still be extracted.

C. Verifiable Secret Sharing

Verifiable Secret Sharing (VSS) scheme allows a dealer to share a secret s among n parties, and s can be recovered from $k+1$ parties. We give the Feldman's VSS scheme [16] below:

- $\mathsf{Setup}(1^\lambda, s)$: For a secret $s \in \mathbb{Z}_q$, the dealer picks some random coefficients $\alpha_1, \ldots \alpha_k \in \mathbb{Z}_q$. Define a polynomial $p(x) = s + \alpha_1 x + \ldots + \alpha_k x^k$. The dealer publishes a set of commitments $\mathbb{C} = (C_0 = g^s, C_1 = g^{\alpha_1}, \ldots, C_k = g^{\alpha_k})$ and stores coefficients $(\alpha_1, \ldots \alpha_k)$.
- $\mathsf{Share}(s, (\alpha_1, \ldots \alpha_k), e_i)$: For a secret s, coefficients $(\alpha_1, \ldots \alpha_k)$ and an index e_i, the dealer sends a share $s_i = p(e_i)$ to party i for $i \in \{1, n\}$.
- $\mathsf{Check}(s', e_i, \mathbb{C})$: Each party i can validate his share by checking if $g^{s'} = \prod_{j=0}^{k} C_j^{e_i^j}$. Output 1 if it is correct.

D. Extended Accountable Assertions

We introduce a new primitive called extended accountable assertion (eAA) to provide message control in blockchain. AA was designed to *prohibit* changes: if there are two asserted messages, the assertion secret key is extracted. On the other hand, we design eAA to *control* changes with the use of VSS: if there are two asserted messages, a relationship value is extracted; and the relationship value is a key share that can be verified in VSS.

Now, we describe the sub-protocol eAA.

- $\mathsf{KeyGen}(1^\lambda, \mathsf{M})$: It takes a security parameter λ and a set of messages M as input, it outputs an assertion public key apk, an assertion secret key ask, and an auxiliary secret key auxsk.
- $\mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m)$: It takes assertion secret keys ask, auxsk and a message m as input, it outputs an assertion τ or $\perp$ for failure ($m \notin \mathsf{M}$).

- $\mathsf{Verify}(\mathtt{apk}, m, \tau)$: It takes an assertion public key $\mathtt{apk}$, a message m, and an assertion τ as input, outputs a bit 1 for valid or 0 for invalid.
- $\mathsf{Extract}(\mathtt{apk}, m, m', \tau, \tau')$: It takes an assertion public key $\mathtt{apk}$, a collided (message, assertion) values $[(m;\tau),(m';\tau')]$ as input, it outputs $f = F(\mathtt{ask}, m, j)$ for some one-way function F, where τ and τ' intersects at position j. Note that f can be verified by $\mathtt{apk}$.

Similar to AA, eAA needs to achieve indistinguishability and collision-resistance. Indistinguishability means that the randomness involved in an assertion does not reveal whether it was obtained from directly hashing or computing them using the assertion key share. Note that an assertion includes a set of (message, randomness) values. Collision-resistance means that an adversary cannot generate assertions on messages when the assertion secret key is unknown.

1) Our eAA Construction with VSS: The proposed eAA works as follows: 1) the Feldman's VSS scheme [16] is used to split an assertion secret key into multiple shares; 2) a share s_i is used as a relationship value and it is extracted from two pairs of collided (message, assertion) values; 3) the extracted share (relationship value) can be checked by the VSS scheme.

Now, we describe VSS and our eAA, which is built on top of AA in [24].

- $\mathsf{KeyGen}(1^\lambda, \mathsf{M})$: It takes a security parameter λ as input, runs $(\mathtt{apk}', \mathtt{ask}', \mathtt{auxsk}') \leftarrow \mathsf{AA.KeyGen}(1^\lambda)$. It runs $(\mathbb{C}, (\alpha_1, \ldots, \alpha_k)) \leftarrow \mathsf{VSS.Setup}(1^\lambda, \mathtt{ask}')$, where $C_0 = \mathtt{apk}' \in \mathbb{C}$. For a message $m \in \mathsf{M}$, it generates a CH public key Z and secret key z. It outputs $\mathtt{ask} = \mathtt{ask}', \mathtt{auxsk} = (\mathtt{auxsk}', (\alpha_1, \ldots, \alpha_k), \{z\}), \mathtt{apk} = (\mathtt{apk}', \{m, Z\})$.
- $\mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m)$: It takes the assertion secret keys $\mathtt{ask}, \mathtt{auxsk}$ and a message $m \in \mathsf{M}$ as input. The context C includes the commitments $\mathbb{C}$ in VSS. It computes $e_i = G(m, C_i)$ for some pseudorandom function G and $C_i \in \mathbb{C}$, and generates a share by $s_i \leftarrow \mathsf{VSS.Setup}(\mathtt{ask}', (\alpha_1, \ldots, \alpha_k), e_i)$ for all i indicating the depth of the tree. It returns $\tau/\perp \leftarrow \mathsf{AA.Assert}(\{s_i\}, \mathtt{auxsk}, C, m)$, where the share s_i is used when computing the nodes in depth i. In particular, z is used as the secret key of CH in the leaf node when running AA.Assert in [24].
- $\mathsf{Verify}(\mathtt{apk}, m, \tau)$: It takes the assertion public key $\mathtt{apk}$, a message m, and an assertion τ as input, outputs $b \leftarrow \mathsf{AA.Verify}(\mathtt{apk}, m, \tau)$. In particular, Z is used as the public key of CH in the leaf node when running AA.Verify in [24].
- $\mathsf{Extract}(\mathtt{apk}, m, m', \tau, \tau')$: It takes the assertion public key $\mathtt{apk}$, a collided (message, assertion) values $[(m;\tau),(m';\tau')]$ as input, outputs $s'/\perp \leftarrow \mathsf{AA.Extract}(\mathtt{apk}, m, m', \tau, \tau')$. If m and m' collides at a node of depth i, observe that $\mathsf{VSS.Check}(s', G(m, C_i), \mathbb{C}) = 1$ for some $C_i \in \mathbb{C}$.

The indistinguishability and collision-resistance of our eAA follows from the indistinguishability and collision-resistance of AA in [24].

2) Message Control Technique: Let M be a set of individual messages. Message-control requires two publicly verifiable conditions: 1) A candidate message $m \in \mathsf{M}$ is verified by a corresponding CH public key Z in $\mathtt{apk}$. 2) Two candidate messages are linked via a relationship value. This condition is achieved using two pairs of collided (message, assertion) values. One is $\tau \leftarrow \mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m)$, the other is $\tau' \leftarrow \mathsf{Assert}(\mathtt{ask}, \mathtt{auxsk}, m')$. Each assertion corresponds to a tree path in the Merkel-style tree (see Figure 2), and a relationship is established if two tree paths have overlap. We show the detailed generation of the relationship value below.

We consider a general case where any candidate message pair can establish a unique relationship. The proposed eAA scheme is designed to handle this case, as the assertion secret key is split into multiple key shares. Each key share may act as a relationship value between two candidate messages. The idea of message-control is to let each generation of an assertion link to a tree path, and use key shares to find collisions from a leaf node to the tree root. If two tree paths have an overlap point, a particular key share can be extracted from two pairs of collided (message, randomness) values. The extracted key share confirms the relationship between two candidate messages, and we denote the extracted key share as a relationship value c. We show how to find various relationships between different message pairs in Figure 2. For example, assertion τ on message m includes three slots in a tree path $(A_{3,4}, A_{2,2}, B_{1,1})$, where $(A_{3,4}, r_{3,4}) = \mathsf{Hash}(Z, m)$. Assertion τ' on message m' in another tree path includes slots $(C_{3,4}, A_{2,2}, B_{1,1})$, where $(C_{3,4}, r'_{3,4}) = \mathsf{Hash}(Z', m')$. One can see that the first overlap point is $A_{2,2} \leftarrow (m, m')$, which means a relationship between two messages is established. In particular, a key share can be extracted by any public user because the key share was used to find the collided (message, randomness) values at slot $A_{2,2}$. Similarly, we can find another relationship for another message pair: $B_{1,1} \leftarrow (m, m'')$.

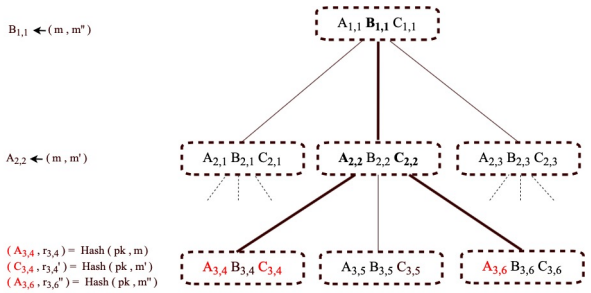

Fig. 2: Assertions in a Merkle-style tree. The tree depth is three, and each tree node has three slots. Slot $A_{2,2}$ indicates a relationship between messages (m, m'), and slot $B_{1,1}$ indicates a relationship between messages (m', m''). Hash means chameleon hash function CH.

If a relationship value is used in a chameleon hash CH, we can achieve the claimed message control. For example, a relationship value c is used to generate a chameleon hash (h, r) on a message $m \in \mathsf{M}$. Later, a trapdoor holder selects a new message $m' \in \mathsf{M}$ and generates the chameleon hash

(h, r') using the same c. As a result, message m links to m' via the relationship value c.

III. Definition and Model

A. Definition

A message-controlled chameleon hash (MCH) consists of the following algorithms.

- $\mathsf{Setup}(1^\lambda)$: It takes a security parameter λ as input, outputs a chameleon key pair $(\mathtt{sk}, \mathtt{pk})$.
- $\mathsf{KeyGen}(\mathtt{pk}, \mathsf{M})$: It takes the chameleon public key $\mathtt{pk}$, and a set of candidate messages $\mathsf{M} = \{m, m'\}$ as input, outputs an assertion key pair $(\mathtt{ask}, \mathtt{apk})$.
- $\mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m)$: It takes the chameleon public key $\mathtt{pk}$, the assertion secret key $\mathtt{ask}$, and a message $m \in \mathsf{M}$ as input, outputs a chameleon hash h, a randomness r, and a relationship value c.
- $\mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m, h, r)$: It takes the chameleon public key $\mathtt{pk}$, assertion public key $\mathtt{apk}$, message m, chameleon hash h and randomness r as input, output a bit $b \in \{0, 1\}$.
- $\mathsf{Adapt}(\mathtt{sk}, \mathtt{ask}, c, m, m', h, r)$: It takes the chameleon secret key $\mathtt{sk}$, the assertion secret key $\mathtt{ask}$, messages m, m', relationship value c, chameleon hash h and randomness r as input, outputs a new randomness r'.

The MCH is *correct* if for all security parameters λ, all keys $(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathsf{Setup}(1^\lambda)$, $(\mathtt{ask}, \mathtt{apk}) \leftarrow \mathsf{KeyGen}(\mathtt{pk}, \mathsf{M})$, for all $m \in \mathsf{M}$, for all $(h, r, c) \leftarrow \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m)$, for all $m' \in \mathsf{M}$, for all $r' \leftarrow \mathsf{Adapt}(\mathtt{sk}, \mathtt{ask}, c, m, m', h, r)$, we have $1 = \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m, h, r) = \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m', h, r')$.

B. Security Models

Indistinguishability. Informally, for a chameleon hash, an adversary cannot decide whether its randomness was freshly generated using Hash algorithm or was created using Adapt algorithm even if the secret keys $(\mathtt{sk}, \mathtt{ask})$ are known. We define a formal experiment in between an adversary $\mathcal{A}$ and a challenger $\mathcal{S}$ Figure 3. The security experiment allows $\mathcal{A}$ to access a HashOrAdapt oracle which ensures that the randomness does not reveal whether it was obtained from Hash or Adapt algorithm. The hashed messages are adaptively chosen by $\mathcal{A}$ from the same message set M.

Experiment $\mathsf{Exp}^{\mathrm{IND}}_{\mathcal{A}}(\lambda)$
$(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathsf{Setup}(1^\lambda), b \leftarrow \{0, 1\}$
$(\mathtt{ask}, \mathtt{apk}) \leftarrow \mathsf{KeyGen}(\mathtt{pk}, \mathsf{M})$
$b' \leftarrow \mathcal{A}^{\mathsf{HashOrAdapt}(\mathtt{sk}, \mathtt{ask}, \cdot, \cdot, b)}(\mathtt{sk}, \mathtt{ask})$
where $\mathsf{HashOrAdapt}(\mathtt{sk}, \mathtt{ask}, \cdot, \cdot, b)$ on input m, m' :
return $\perp$, if $m, m' \notin \mathsf{M}$ or $c = \perp$
$(h_0, r_0, c) \leftarrow \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m')$
$(h_1, r_1, c) \leftarrow \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m)$
$r_1 \leftarrow \mathsf{Adapt}(\mathtt{sk}, \mathtt{ask}, c, m, m', h_1, r_1)$
return (h_b, r_b, c)
return $\perp$ if $r_0 = \perp \vee r_1 = \perp$
return 1, if $b' = b$; else, return 0.

Fig. 3: Indistinguishability.

We require that $\mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m', h_0, r_0) = \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m, h_1, r_1) = 1$, and we define the advantage of the adversary as

$$\mathtt{Adv}^{\mathrm{IND}}_{\mathcal{A}}(\lambda) = |\Pr[\mathsf{Exp}^{\mathrm{IND}}_{\mathcal{A}}(\lambda) \to 1] - 1/2|.$$

Definition 2: A MCH scheme is indistinguishable if for any probabilistic polynomial-time (PPT) $\mathcal{A}$, $\mathtt{Adv}^{\mathrm{IND}}_{\mathcal{A}}(\lambda)$ is negligible in λ.

Collision Resistance. Informally, an adversary attempts to find collisions without using relationship values. The model is formalized by an Adapt' oracle which always inputs a message m' such that $m, m' \in \mathsf{M}$ and c is established between (m, m'). The goal of $\mathcal{A}$ is to output a collision r' associated with a message m' on a non-adversarially generated chameleon hash (h, m, r), while the collision r' was not previously generated by the Adapt' oracle. We define a formal experiment in Figure 4.

Experiment $\mathsf{Exp}^{\mathrm{CR}}_{\mathcal{A}}(\lambda)$
$(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathsf{Setup}(1^\lambda), (\mathtt{ask}, \mathtt{apk}) \leftarrow \mathsf{KeyGen}(\mathtt{pk}, \mathsf{M}), \mathcal{Q} \leftarrow \emptyset$
$(m^*, r^*, m'^*, r'^*, h^*) \leftarrow \mathcal{A}^{\mathsf{Hash}', \mathsf{Adapt}'}(\mathtt{pk}, \mathtt{apk})$
where $\mathsf{Hash}'(\mathtt{pk}, \mathtt{ask}, \cdot)$ on input m :
$(h, r, c) \leftarrow \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m)$
return $\perp$, if $h = \perp$
let $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(\mathtt{pk}, h, m)\}$
return (h, r, c)
where $\mathsf{Adapt}'(\mathtt{sk}, \mathtt{ask}, \cdot, \cdot, \cdot, \cdot, \cdot)$ on input m, m', h, r, c :
return $\perp$, if $(\mathtt{pk}, h, m, \cdots) \notin \mathcal{Q}$
return $\perp$, if $m, m' \notin \mathsf{M}$ or $c = \perp$
$r' \leftarrow \mathsf{Adapt}(\mathtt{sk}, \mathtt{ask}, c, m, m', h, r)$
return r'
return 1, if $1 = \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m^*, h^*, r^*)$
$= \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, m'^*, h^*, r'^*) \wedge (\mathtt{pk}, h^*, \cdot) \in \mathcal{Q}$
$\wedge (\mathtt{pk}, h^*, m^*, \cdot) \notin \mathcal{Q} \wedge m^* \neq m'^*$
else, return 0.

Fig. 4: Collision-Resistance.

We define the advantage of the adversary as

$$\mathtt{Adv}^{\mathrm{CR}}_{\mathcal{A}}(\lambda) = \Pr[\mathsf{Exp}^{\mathrm{CR}}_{\mathcal{A}}(\lambda) \to 1].$$

Definition 3: A MCH scheme is collision resistant if for any PPT $\mathcal{A}$, $\mathtt{Adv}^{\mathrm{CR}}_{\mathcal{A}}(\lambda)$ is negligible λ.

IV. Construction and Analysis

A. An Envisioned Solution

We consider a simple two-party setting: a content owner and an authorized modifier. The content owner publishes its original transaction that includes a set of acceptable values and the authorized modifier's public key. Later, the modifier selects an appropriate value from the set of acceptable values, signs the value with his/her secret key, and updates the transaction with the selected value. Once the updated transaction is included in the blockchain, it represents the new state of the application while the original transaction remains in the blockchain.

If the modifier selects an *arbitrary* value which is not from the set of acceptable values, and signs the value with his/her secret key, the updated transaction will not be accepted by the blockchain. If such updated transactions are submitted to

the blockchain, all participants including miners can verify that they contain *unacceptable* values, and ignore such transactions. Using our proposed MCH, we can ensure that only updated transactions that contain no unacceptable values are accepted into the blockchain.

B. Overview of Our Solution

We provide an overview of MCH and certain design rationales. We define a blockchain system with n users, and we use "transaction owner" (e.g., payer) and "transaction modifier" (e.g., payee) to present MCH. The MCH is based on two chameleon hash functions: CH_0 and CH_1. CH_0 is a semi-public-coin chameleon hash on a message m, where a relationship value c is used to calculate the secret key of the chameleon hash. CH_1 is an extractable chameleon hash for message control (i.e., a simplified version of eAA). Let $(\mathtt{ask}, \mathtt{apk}) \leftarrow \mathsf{KeyGen}_1(1^\lambda)$ be a chameleon key pair. CH_1 has an extra algorithm: $\mathsf{Extract}_1(\mathtt{apk}, h, m, m', r, r')$, which takes the chameleon public key $\mathtt{apk}$, two pairs of collided (message, randomness) values $[(m; r), (m'; r')]$ as input, outputs a share s'. Here, we denote the extracted share as the relationship value c.

We let a transaction owner and a transaction modifier share a key pair $(\mathtt{ask}, \mathtt{apk})$, and the modifier's chameleon public key $\mathtt{pk}_m$ is known to the owner. The transaction owner publishes a candidate message set M. To hash a message $m \in \mathsf{M}$, he picks a tree depth i^* and computes a relationship value c using $\mathsf{VSS.Share}$ on i^* in eAA. A chameleon public key $\mathtt{pk}'$ is computed from $\mathtt{pk}_m$ and c. the transaction owner outputs a chameleon hash tuple: (h_0, h_1, m, r_0, r_1), where $(h_0, r_0) = \mathsf{Hash}_0(\mathtt{pk}', m)$, $(h_1, r_1) = \mathsf{Hash}_1(\mathtt{apk}, m)$.

Upon receiving a chameleon hash tuple, the transaction modifier outputs a new chameleon hash tuple: $(h_0, h_1, m', r'_0, r'_1)$, where $r'_0 = \mathsf{Adapt}_0(\mathtt{sk}', m, m', h_0, r_0)$, and $r'_1 = \mathsf{Adapt}_1(\mathtt{ask}, u, m', h_1, v)$. Note that the modifier uses her chameleon secret key $\mathtt{sk}$ and the relationship value c to generate $\mathtt{sk}'$ and to find a new randomness r'_0. In particular, given two pairs of collided (message, randomness) values, any public user can extract the secret key $\mathtt{ask} = c$ due to the CH_1's extractability, i.e., $\mathtt{ask} \leftarrow \mathsf{Extract}_1(\mathtt{apk}, h_1, m, m', r_1, r'_1)$. Then, the public verify the chameleon hash: $h_0 = \mathsf{Hash}_{\mathtt{pk}'}(m; r_0) = \mathsf{Hash}_{\mathtt{pk}'}(m'; r'_0)$, which indicates that the relationship between messages (m, m') is established.

To resist key-exposure problem regarding CH_0 [9] (namely, the secret trapdoor $\mathtt{sk}$ can be publicly extracted given a collision for hash h_0), the transaction owner hides the randomness r_0 using public key encryption scheme PKE. Specifically, PKE takes the chameleon public key $\mathtt{pk}_m$ and randomness r_0 as input, outputs a ciphertext C. The intention is to let transaction modifier decrypt the randomness r_0 and find a new randomness r'_0. The transaction modifier uses the same technique to hide the randomness r'_0.

C. Construction

The proposed construction consists of the following building blocks.

- An extended accountable assertion scheme $\mathsf{eAA} = (\mathsf{KeyGen}, \mathsf{Assert}, \mathsf{Verify}, \mathsf{Extract})$.
- A semi-public-coin chameleon hash scheme $\mathsf{CH} = (\mathsf{KeyGen}, \mathsf{Hash}, \mathsf{Verify}, \mathsf{Adapt})$.
- A public key encryption scheme $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.

We assume that a transaction owner hashes a message m, while a transaction modifier replaces it by a new message m'.

- $\mathsf{Setup}(1^\lambda)$: It generates a CH key pair $(\mathtt{sk}_c, \mathtt{pk}_c) \leftarrow \mathsf{CH.KeyGen}(1^\lambda)$ and an encryption key pair $(\mathtt{sk}_e, \mathtt{pk}_e) \leftarrow \mathsf{PKE.KeyGen}(1^\lambda)$ for the transaction modifier. It outputs $\mathtt{pk}_m = (\mathtt{pk}_c, \mathtt{pk}_e)$ and $\mathtt{sk}_m = (\mathtt{sk}_c, \mathtt{sk}_e, \mathtt{pk}_m)$.
- $\mathsf{KeyGen}(\mathtt{pk}_m, \mathsf{M})$: A transaction owner outputs an assertion key pair $(\mathtt{ask}', \mathtt{auxsk}', \mathtt{apk}) \leftarrow \mathsf{eAA.KeyGen}(1^\lambda, \mathsf{M})$. It outputs $\mathtt{apk}$, $\mathtt{ask} = (\mathtt{ask}', \mathtt{auxsk}', \mathtt{apk})$. In addition, the ciphertext $C_{\mathtt{ask}} \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \mathtt{ask})$ is sent to the transaction modifier.
- $\mathsf{Hash}(\mathtt{pk}_m, \mathtt{ask}, m)$: The transaction owner takes a message $m \in \mathsf{M}$ as input, he computes $\tau \leftarrow \mathsf{eAA.Assert}(\mathtt{ask}', \mathtt{auxsk}', m)$.
  He picks a tree depth i^* and computes a share e_{i^*} using $\mathsf{VSS.Share}$ in eAA. He comptues a CH public key $\mathtt{pk}'$ using $\mathtt{pk}_c$ and e_{i^*}. Then he computes $(h, \xi) \leftarrow \mathsf{CH.Hash}(\mathtt{pk}', m; \rho)$ using a randomness ρ, and computes the ciphertext $C_\rho \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho)$. He outputs a hash h, a randomness $r = (\xi, C_\rho, \tau, \mathtt{pk}')$ and a relationship value $c = e_{i^*}$.
- $\mathsf{Verify}(\mathtt{pk}_m, \mathtt{apk}, m, h, r = (\xi, C_\rho, \tau, \mathtt{pk}'))$: One can verify whether a given hash h is valid. Output 1 if: $1 \leftarrow \mathsf{CH.Verify}(\mathtt{pk}_c, m, h, \xi)$ and $1 \leftarrow \mathsf{eAA.Verify}(\mathtt{apk}, m, \tau)$. Otherwise output 0.
- $\mathsf{Adapt}(\mathtt{sk}_m, \mathtt{ask}, c, m, m', h, r = (\xi, C_\rho, \tau, \mathtt{pk}')$: The transaction modifier with a chameleon secret key $\mathtt{sk}$, a message $m' \in \mathsf{M}$, and the relationship value $\mathtt{sk}_c$ performs the following operations
  1) Verify $1 \leftarrow \mathsf{Verify}(\mathtt{pk}_m, \mathtt{apk}, m, h, r)$.
  2) Compute $\tau' \leftarrow \mathsf{eAA.Assert}(\mathtt{ask}', \mathtt{auxsk}', m')$.
  3) Compute a CH secret key $\mathtt{sk}'$ by using the share e_{i^*} and $\mathtt{sk}_c$. Compute $(\rho', \xi') \leftarrow \mathsf{CH.Adapt}(\mathtt{sk}', m, m', \rho, h)$, where $\rho \leftarrow \mathsf{PKE.Dec}(\mathtt{sk}_e, C_\rho)$. Compute the ciphertext $C_{\rho'} \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho')$.
  4) Output $(h, r = (\xi', C'_\rho, \tau', \mathtt{pk}'), c)$, where $C_{\rho'} \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho')$.

Correctness. Given two pairs of collided (message, assertion) values (m, τ) and (m', τ') that corresponds to the same h, any public user can extract an assertion key share $s' \leftarrow \mathsf{eAA.Extract}(\mathtt{apk}, m, m', \tau, \tau')$, verify the chameleon hash $\mathsf{CH.Verify}(\mathtt{pk}_c, m, h, \xi) = \mathsf{CH.Verify}(\mathtt{pk}_c, m', h, \xi') = 1$. The public verification requires that both the owner and the modifier generate assertions (τ, τ'). If the modifier avoids

using the assertion key share s' to generate assertion τ', the relationship between two messages cannot be publicly verified.

Remark. One may notice that the transaction modifier may generate a fake (or invalid) assertion τ^* using a candidate message m', and output $(h, m', r' = (\xi', C_{\rho'}, \tau^*, \mathtt{pk}'))$. If the candidate message m' appears in the blockchain, then the rewriting on the transaction succeeds as $1 \leftarrow \mathsf{CH.Verify}(\mathtt{pk}_c, m', h, \xi')$. However, this modified transaction becomes "invalid" since $1 \neq \mathsf{eAA.Verify}(\mathtt{apk}, m', \tau^*)$; thus, this transaction cannot be accepted by the blockchain participants. This attack by the transaction modifier is possible. We argue that the transaction modifier's best interest is to maliciously modify the hashed messages using the arbitrarily-chosen ones, but not to invalidate the mutable transactions in the blockchain. One may also notice that the modifier can rewrite a mutable transaction multiple times. To control the rewriting times, one can use double-authentication-preventing signatures (DAPS) [12], [22] to construct MCH, and we leave this as a future work.

D. Security Analysis

We show the security result of our proposed construction in the form of the following theorems. The detailed proofs are given in Appendix A.

Theorem 4: The MCH scheme is indistinguishable if eAA and semi-public-coin CH are indistinguishable.

Theorem 5: The MCH scheme is collision resistant if eAA and semi-public-coin CH are collision resistant.

V. Instantiation and Evaluation

Let $\mathbb{G}$ denote a cyclic group with prime order q and generator g. Let $\mathtt{H}_0 : \{0,1\}^* \rightarrow \mathbb{Z}_q$, and $\mathtt{H}_1 : \{0,1\}^* \rightarrow \mathbb{G}$ be collision resistant hash functions. Let F_κ be a pseudo-random function, where κ is a secret key.

Firstly, all candidate messages can be converted from bit-strings to values in $\mathbb{Z}_q$ by a pseudorandom random function $\mathcal{F}_z : \mathcal{M} \times \mathcal{K} \rightarrow \mathbb{Z}_q$ for some key z in a key space $\mathcal{K}$ [7]. We denote that the message set M contains the converted messages in $\mathbb{Z}_q$.

Then, we instantiate the underlying cryptographic building blocks. First, to instantiate the eAA scheme, we use accountable assertion AA [24] and Feldman's VSS [16] to construct eAA. The proposed eAA scheme is embedded into the instantiation. Specifically, both Hash and Adapt algorithms include an assertion generation, and Verify algorithm includes an assertion verification. In particular, any pubic user can run the extraction process to ensure the correctness of message-control. Second, we use the DL-based chameleon hash [11], [19] to instantiate the underlying CH scheme. Meanwhile, the RSA-based and DL-based chameleon hash equipped with bilinear map [11] should also be suitable. Third, the PKE protocol can be instantiated to ElGamal encryption scheme [15].

A. Instantiation

- $\mathsf{Setup}(1^\lambda)$: It takes a security parameter λ as input, the transaction modifier outputs a chameleon key pair $(\mathtt{sk}_c, \mathtt{pk}_c)$, where $\mathtt{sk}_c \in \mathbb{Z}_q^*$, $\mathtt{pk}_c = g^{\mathtt{sk}_c}$. It generates an ElGamal encryption key pair $(\mathtt{sk}_e, \mathtt{pk}_e = g^{\mathtt{sk}_e})$ for the transaction modifier. It outputs $\mathtt{pk}_m = (\mathtt{pk}_c, \mathtt{pk}_e)$ and $\mathtt{sk}_m = (\mathtt{sk}_c, \mathtt{sk}_e, \mathtt{pk}_m)$.
- $\mathsf{KeyGen}(\mathtt{pk}_m, \mathsf{M})$: A transaction owner chooses a pseudo-random key $\kappa \leftarrow \{0,1\}^\lambda$, and publishes a Merkle-style tree Λ with depth k. Each tree node includes a number of dummy (message, randomness) values: $\{(u_j^i, v_j^i)\}^{n_0}$, where $u_j^i = \mathsf{F}_\kappa(p, a_j, 0)$, $v_j^i = \mathsf{F}_\kappa(p, a_j, 1)$, $i \in \{1, k\}$ is the depth of the node, $j \in \{1, n_0\}$ is the index inside the node, p denotes the unique identifier for the position of a tree node in Λ, a_j denotes the position of u_j^i inside a tree node, and n_0 denotes the number of dummy messages in a tree node. Meanwhile, it maintains an initially empty set L of *used* leaf positions. The transaction owner runs $\mathsf{eAA.KeyGen}(1^\lambda, \mathsf{M})$ to obtain $\mathtt{apk}'$, $\mathtt{ask}'$ and $\mathtt{auxsk}'$. It outputs $\mathtt{apk} = (\mathtt{apk}', \Lambda)$, $\mathtt{ask} = (\mathtt{ask}', \mathtt{auxsk}', \kappa, \mathtt{apk})$. The ciphertext $C_{\mathtt{ask}} \leftarrow \mathsf{ElGamal.Enc}(\mathtt{pk}_e, \mathtt{ask})$ is sent to the transaction modifier.
- $\mathsf{Hash}(\mathtt{pk}_m, \mathtt{ask}, m)$: To hash a candidate message m at a leaf node Y_k of depth k, recall that $\mathtt{ask} = (\mathtt{ask}', \mathtt{auxsk}', \kappa, \mathtt{apk})$ and:
  - the dummy (message, randomness) values are in the form of $(u_j^i, v_j^i) = [\mathsf{F}_\kappa(p, a_j, 0), \mathsf{F}_\kappa(p, a_j, 1)]$, where a_j denotes the position of u_j^i within Y_i.
  - for the VSS scheme, $\mathtt{ask}'$ is shared by the polynomial $f(x) = \mathtt{ask}' + \sum_{\ell=1}^{k} \alpha_\ell \cdot x^\ell$ for some $\alpha_\ell \in \mathtt{auxsk}'$. The commitments $C_i = g^{\alpha_i}$ are used to check the validity of the shares. Denote $F(x) = g^{f(x)}$.
  - since $m \in \mathsf{M}$, there is a corresponding value $z \in \mathtt{auxsk}'$.

  The transaction owner performs the following operations. Firstly, we show the generation of an assertion τ. The transaction owner computes an assertion path $(Y_k, a_k, Y_{k-1}, a_{k-1}, \cdots, Y_1, a_1)$ from a leaf Y_k to the root Y_1.
  1) Computing $y_{a_k}^k$: Compute dummy values $(u_{a_k}^k, v_{a_k}^k)$ and $e_k = \mathtt{H}_0(m, C_k, a_k)$. Assert the message m to leaf node Y_k by computing the collision $v_{a_k}^{'k} = v_{a_k}^k + (u_{a_k}^k - m) \cdot z/f(e_k)$. The message m links to a slot a_k in Y_k, so the transaction owner adds a_k to L. Compute the entry $y_{a_k}^k = \mathtt{H}_1(g^{z \cdot m} \cdot g^{f(e_k) \cdot v_{a_k}^{'k}}, v_{a_k}^{'k})$.
  2) Compute the remaining entries $y_j^k = g^{u_j^k} \cdot g^{f(e_k) \cdot v_j^k}$ for $j \in \{1, n_0\} \setminus \{a_k\}$. The leaf Y_k stores the entries $(y_1^k, \cdots, y_{n_0}^k)$.
  3) Set an intermediate value $Z_k = \mathtt{H}_0(y_1^k, \cdots, y_{n_0}^k)$, and $\zeta_k = (y_1^k, \cdots, y_{a_{k-1}}^k, F(e_k), y_{a_{k+1}}^k \cdots, y_{n_0}^k)$.

  Next, it computes the remaining nodes up to the root for $w = k-1, \cdots, 1$ as follows.
  1) Assert the intermediate value Z_{w+1} to node Y_w by computing $v_{a_w}^{'w} = v_{a_w}^w + (u_{a_w}^w - Z_{w+1})/f(e_w)$, where $e_w = \mathtt{H}_0(u_{a_w}^w, C_w, a_w)$. Compute the entry $y_{a_w}^w = g^{Z_{w+1}} \cdot g^{f(e_w) \cdot v_{a_w}^{'w}}$.

2) Compute the remaining entries $y_j^w = g^{u_j^w} \cdot g^{f(e_w)\cdot v_j^w}$ for $j \in \{1, n_0\}\backslash\{a_w\}$. The leaf Y_w stores the entries $(y_1^w, \cdots, y_{n_0}^w)$.
3) Set $Z_w = \mathtt{H}_0(y_1^w, \cdots, y_{n_0}^w)$, and $\zeta_w = (y_1^w, \cdots, y_{a_{w-1}}^w, F(e_w), y_{a_{w+1}}^w \cdots, y_{n_0}^w)$.

The assertion on message m is $\tau = \{(v_{a_k}^{'k}, \zeta_k, a_k), \cdots, \underline{(v_i^{'i}, \zeta_i, a_i)}, \cdots, (v_1^{'1}, \zeta_1, a_1), \underline{Z_1}\}$. Note that the component ζ_i includes the value $F(e_i) = g^{f(e_i)}$. The assertion public key in the root node Y_1 is computed as $Z_1 = \mathtt{H}_0(y_1^1, \cdots, y_{n_0}^1)$.

Denote the index $i^* \in [1, k]$ which allows modification and compute key share $c = f(e_{i^*})$ and $\mathtt{pk}' = g^c$. Pick a random $\rho \in \mathbb{Z}_q^*$ and compute chameleon hash $h = g^{c\cdot m} \cdot \mathtt{pk}_c^{\rho}$, $\xi = (R = \mathtt{pk}_c^{\rho}, \pi)$, where π is the ZK proof of ρ. Encrypt ρ by $C_\rho \leftarrow \mathsf{ElGamal.Enc}(\mathtt{pk}_e, \rho)$. It outputs a hash h, a randomness $r = (\xi, C_\rho, \tau, \mathtt{pk}')$ and a relationship value c.

- $\mathsf{Verify}(\mathtt{pk}_m, \mathtt{apk}, m, h, r = (\xi, C_\rho, \tau, pk'))$: One can verify whether a given chameleon hash h is valid or not. The algorithm outputs 1, if the following conditions hold:
  1) Compute $e_w = \mathtt{H}_0(m, C_w, a_w)$ and verify the key shares $F(e_w)$ for w$= 1, \ldots, k$:
  $$F(e_w) \stackrel{?}{=} g^{\mathtt{ask}'} \cdot C_1^{e_w} \cdot C_2^{e_w^2} \cdots C_w^{e_w^w}.$$
  2) Parse $\xi = (R, \pi)$. Verify the chameleon hash $h \stackrel{?}{=} F(e_{i^*})^m \cdot R$ and π is a valid ZK proof for $\mathtt{pk}_c$.
  3) For level k, compute $y_{a_k}^k = \mathtt{H}_1((g^z)^m \cdot F(e_k)^{v_{a_k}^{'k}}, v_{a_k}^{'k})$ and $Z_k = \mathtt{H}_0(y_1^k, \cdots, y_{n_0}^k)$ (the remaining inputs y_i^k are from ζ_k). Compute recursively, for level $w = k-1, \ldots 1$, that $y_{a_w}^w = g^{Z_{w+1}} \cdot F(e_w)^{v_{a_w}^{'w}}$ and $Z_w = \mathtt{H}_0(y_1^w, \cdots, y_{n_0}^w)$. Finally check if the assertion public key is equal to Z_1.
- $\mathsf{Adapt}(\mathtt{sk}_m, \mathtt{ask}, c = f(e_{i^*}), m, m', h, r = (\xi, C_\rho, \tau, \mathtt{pk}'))$: A transaction modifier performs the following operations
  1) Verify the chameleon hash tuple as described above.
  2) Generate a new assertion τ' on the message m' using the assertion secret key $\mathtt{ask}$. Specifically, the transaction modifier performs the following steps to assert the message m' to a node Y_k.
     a) Compute the collision $v_{a_k}^{'k} = v_{a_k}^k + (u_{a_k}^k - m') \cdot z'/f(e_k')$, where $f(e_k') = \mathtt{ask}' + \sum_{\ell=1}^{k} \alpha_\ell \cdot e_k^{'\ell}$, $e_k' = \mathtt{H}_0(m', C_k, a_k)$. In addition, the transaction modifier adds a_k to L.
     b) Compute the entry $y_{a_k}^k = \mathtt{H}_1(g^{z'\cdot m'} \cdot g^{f(e_k')\cdot v_{a_k}^{'k}}, v_{a_k}^{'k})$.
     c) Compute the remaining entries $y_j^k = g^{u_j^k} \cdot g^{f(e_k')\cdot v_j^k}$ for $j \in \{1, n_0\}\backslash\{a_k\}$. The leaf Y_k stores the entries $(y_1^k, \cdots, y_{n_0}^k)$.
     d) Set $Z_k = \mathtt{H}_0(y_1^k, \cdots, y_{n_0}^k)$, and $f_k = (y_1^k, \cdots, F(e_k'), \cdots, y_{n_0}^k)$.
     e) Compute the remaining nodes up to the root for $w = k-1, \cdots, 1$ using the same method as described in the Hash algorithm. For example, the transaction modifier uses a key share $f(e_w)$ for $e_w = \mathtt{H}_0(u_j^w, C_w, a_j)$ to assert an intermediate value Z_{w+1} to node Y_w. Eventually, the assertion on message m' is $\tau' = \{(v_{a_k}^{'k}, f_k, a_k), \cdots, \underline{(v_i^{'i}, f_i, a_i)}, \cdots, (v_1^{'1}, f_1, a_1), Z_1\}$.
  3) Compute a new randomness as $\rho' = \rho + (m - m')\cdot c/\mathtt{sk}_c$, where $\rho \leftarrow \mathsf{ElGamal.Dec}(\mathtt{sk}_e, C_\rho)$, $c = f(e_{i^*})$.
  4) Compute the value $\xi' = (R' = \mathtt{pk}_c^{\rho'}, \pi')$ where π' is a ZK proof for ρ' and compute an ElGamal ciphertext $C_{\rho'} \leftarrow \mathsf{ElGamal.Enc}(\mathtt{pk}_e, \rho'))$.
  5) Output $(h, r' = (\xi', C_{\rho'}, \tau', \mathtt{pk}'), c)$.

Correctness. The transaction owner uses a secret value z to compute a collision when he asserts a message m to a leaf node Y_k. This technique is chameleon hash with ephemeral trapdoor [11]. As a consequence, one can check $F(e_k)^{v_{a_k}^{'k}} \cdot g^{z\cdot m} \stackrel{?}{=} F(e_k)^{v_{a_k}^k} \cdot g^{z\cdot u_{a_k}^k}$ given $(g^z, F(e_k))$. Similarly, the transaction modifier computes a collision using a secret value z' (not random one) when she asserts a candidate message m' to a leaf node Y_k. The purpose is to prevent the transaction modifier from choosing arbitrary message to generate an assertion.

If a key share $f(e_{i^*})$ is used in two assertions (τ, τ'), any public user can extract it by computing $f(e_{i^*}) = (Z_{i+1} - Z'_{i+1})/(v_j^{''i} - v_j^{'i})$. Note that the randomness $(v_j^{'i}, v_j^{''i})$ are adapted from the same dummy randomness v_j^i. Specifically, the key share $f(e_{i^*})$ is used to find collisions at level i for (τ, τ'), such that $g^{Z_{i+1}} \cdot g^{f(e_{i^*})\cdot v_j^{'i}} = g^{Z'_{i+1}} \cdot g^{f(e_{i^*})\cdot v_j^{''i}}$, the intermediate values (Z_{i+1}, Z'_{i+1}) are associated with candidate messages (m, m') respectively. The public can verify the chameleon hash via the following equation

$$\begin{aligned} h &= g^{f(e_{i^*})\cdot m'} \cdot g^{\mathtt{sk}_c\cdot\rho'} \\ &= g^{f(e_{i^*})\cdot m'} \cdot g^{\mathtt{sk}_c\cdot[\rho + (m-m')f(e_{i^*})/\mathtt{sk}_c]} \\ &= g^{f(e_{i^*})\cdot m} \cdot g^{\mathtt{sk}_c\cdot\rho} \end{aligned}$$

B. Evaluation

We evaluate the performance of MCH for blockchain rewriting based on a proof-of-concept implementation in Java. On a high-level, three schemes are constructed. Scheme 1 creates an immutable blockchain system with basic functionalities and a PoW consensus mechanism [17]. The system is specifically designed to include ten blocks, and each block consists of 100 transactions (one can extend it to real-world transactions such as a block containing 3500 transactions). We generate ten threads to simulate ten nodes in a peer-to-peer network, and each thread can be regarded as a single node. All these threads are managed by a modular for broadcasting transactions and new mined blocks to all nodes. A chain of blocks is established with PoW by consolidating transactions broadcast by the ten nodes. The code of our proof-of-concept implementation is available on GitHub [6].

Scheme 2 is similar to Scheme 1, except that the proposed MCH is applied to certain mutable transactions (to be precise, some of the mutable contents present in these transactions) in Scheme 2. A randomly-chosen node serves as the authority (e.g., KGC in [13], [29]) to rewrite the blockchain, while the remaining nodes are allowed to contribute to the consensus

in Scheme 2. Each mutable transaction is generated by its transaction owner using the authority's chameleon public key; the authority who holds the corresponding chameleon secret key can perform rewriting before its time-lock expires. The authority may distribute its rewriting permissions to certain other nodes. The authority also generates a Merkle-style tree as described in the Figure 2.

Scheme 3 is similar to Scheme 2, except that no authority is needed in Scheme 3. The performance of Scheme 3 is also similar to that of Scheme 2 because the underlying primitives are nearly identical. Therefore, we focus on the evaluation of Scheme 1 and Scheme 2 only.

C. Benchmarking

Our benchmarking experiments are performed on a desktop computer with Inter Core i7 at 2.9GHz, and 16GB RAM. We choose a hash function SHA-256 and an ElGamal group of 128 bit-length order [15]. In Scheme 2, if there is no mutable transaction in a block, then the block is formed and validated in the same way as in Scheme 1. We thus focus on the evaluation of mutable transactions in Scheme 2.

We first evaluate the performance of MCH which includes the evaluation of: Hash, Verify, Adapt algorithms and correctness check (i.e., checking the relationship between two (message, randomness) values with the same chameleon hash output). Then, we measure the running time for generating a Merkle tree on a set of transactions (including mutable ones) and for validating blocks in PoW. Each mutable transaction is set to contain two candidate messages in our evaluation.

Figure $5(a)$ shows that the running time of Hash algorithm ranges between 0.8 ms and 2.0 ms as the depth of the tree changes from five to ten, and the number of dummy messages in each tree node varies from six to ten. Figures $5(b)$–(d) show similar trends for the running time of Verify, Adapt, and correctness check, respectively.

Table II shows the performance comparison between Scheme 1 and Scheme 2 in terms of transaction generation time, transaction overhead, Merkle tree generation time, and PoW validation time. We set ten mutable transactions in each block, which means 10% of the transactions are mutable according to [14], and each transaction is assumed to contain one mutable content. We also set each mutable transaction to contain two candidate messages and tree parameters $k = 10$ and $n_0 = 10$. Let $\mathcal{L}_{\mathsf{CH}}$ denote the length of chameleon hash CH, which includes a chameleon hash $\mathcal{L}_{\mathsf{CH}_h}$ and a randomness $\mathcal{L}_{\mathsf{CH}_\rho}$. Let the size of (plain) candidate messages associated with a mutable transaction be $\mathcal{L}_{\mathsf{M}}$. We denote the length of eAA and PKE as $\mathcal{L}_{\mathsf{eAA}}$ and $\mathcal{L}_{\mathsf{PKE}}$, respectively.

From Table II, we observe that the generation of a mutable transaction in Scheme 2 takes approximately 2 ms, which is longer than 0.002 ms in Scheme 1. On the other hand, the storage overhead of a mutable transaction in Scheme 2 is higher than a normal transaction in Scheme 1. This is due to the fact that additional cryptographic primitives, including eAA and PKE, are used in Scheme 2. These primitives incur additional computational and storage costs, which depend on the structure of the assertion tree and the number of candidate messages involved in each mutable transaction. The performance of Merkle tree generation and PoW validation in Scheme 2 is nearly identical to that in Scheme 1, which confirms that the proposed MCH can be easily integrated into the existing blockchain systems.

TABLE II: The performance comparison between Scheme 1 and Scheme 2. Tx (time) denotes the time to generate a transaction, and Tx (overhead) denotes the transaction's storage cost. Symbol $*$ includes $\mathcal{L}_{\mathsf{CH}_h} + \mathcal{L}_{\mathsf{eAA}} + 2\mathcal{L}_{\mathsf{PKE}} + \mathcal{L}_{\mathsf{M}}$. $2\mathcal{L}_{\mathsf{PKE}}$ means one encryption is performed on an assertion secret key `ask`, and one encryption is performed on a randomness ρ.

	Tx (time)	Merkle	PoW	Tx (overhead)
Scheme 1:	0.002 ms	75710 ns	24 ns	256-bit
Scheme 2:	2.04 ms	73935 ns	22 ns	256-bit + $*$

VI. Integrating MCH into Blockchains

In this section, we show how the proposed message-controlled chameleon hash (MCH) can be integrated into a blockchain. Specifically, we consider an immutable blockchain in the permissioned setting (where certain permission is required to access the blockchain) as well as in the permissionless setting (where anyone can join the network as a node and access the blockchain). In addition, we address the issue regarding transaction inconsistency in the blockchain [14], [23], when the monetary value involved in a transaction already present in the blockchain needs to be modified. We consider the payer of a transaction to be the *transaction owner*. On the other hand, either the payee or the payer can act as a *transaction modifier*.

The idea of restricted modification is the following: one of the parties (i.e., payer/payee) acts as the transaction modifier and the other one chooses the set of permissible messages — which prevents each party from rewriting a transaction arbitrarily. There is a wide range of applications that will benefit from rewriting a transaction. Certain examples, where the payee acts as the transaction modifier and the payer chooses the set of messages, include cashback on shopping using credit cards, rebate for timely payment of taxes. On the other hand, there are certain examples where the payer is the transaction modifier. Transactions made towards provident funds are of this type as the payer (e.g., an employee) would want to transfer more money due to higher interest rates, but the payee (e.g., government managing such funds) would put restrictions on the range of a transaction (to be more precise, the payee fixes the set of possible messages that the payer is allowed to use in order to replace the original transaction amount and thus imposes an upper bound on the possible transaction amount).

A. Transaction Inconsistency

MCH is designed to modify data present in the blockchain in a controlled manner. The types of modification could be various depending on the application scenarios. For example, one might want to modify the monetary amount of a

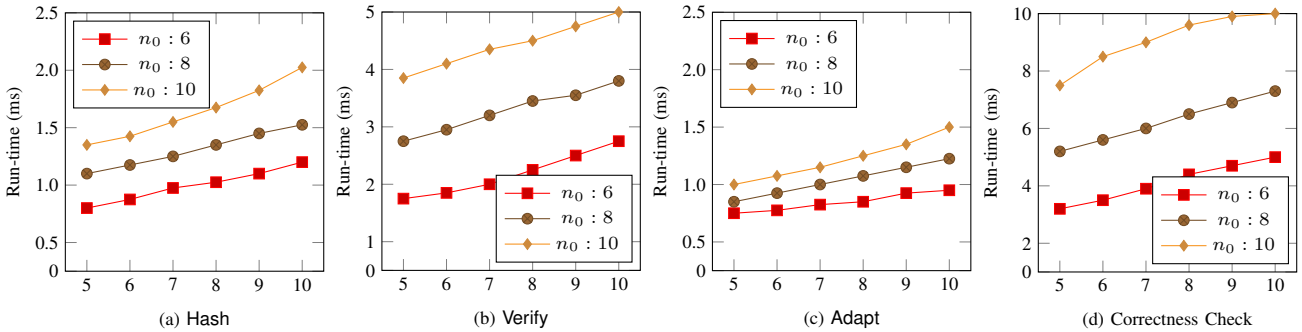

Fig. 5: Run-time of Hash, Verify, Adapt algorithms and correctness check, where n_0 denotes the number of dummy messages in a tree node, and the running time varying with respect to the depth of tree $k \in \{5, 10\}$.

transaction already registered in the blockchain. However, this can potentially result in transaction inconsistency in the blockchain [14], [23]. This is due to the fact that modifying the amount of a transaction may affect the subsequent transactions present in the blockchain. We note that modifying a transaction must not affect the transaction fee associated with it. So, the set of messages must be chosen in a such a way that the transaction fee remains unchanged.

We address transaction inconsistency due to changes in the transaction amount as follows. The transaction modifier is not allowed to modify a transaction that is already spent; otherwise it would require tracing all the subsequent transactions (in a way similar to taint analysis in software security) and changing some of them (which may cause a dispute among multiple parties — we leave this as a future work). To achieve our goal, a time-lock [1] is embedded in each mutable transaction using a timestamp T [3] that is agreed upon and signed by both the payer and the payee (to simplify our description, we focus on the case in which each transaction is associated with a single payer and payee; it is not difficult to extend the case for multiple payers and payees). The modification on the transaction amount must be performed prior to the specified timestamp T. In other words, the payee cannot spend this mutable transaction before time T, and the transaction becomes unchangeable and spendable once T is reached; therefore, monetary transactions can only be rewritten when they are in the *unspent* state. Suppose the blockchain includes a valid transaction Tx with a time-lock T. The transaction modifier later may wish to modify the monetary value of the transaction within time $\mathsf{T} - \delta$, where δ is the time elapsed between a transaction getting broadcasted and the transaction getting included in the blockchain, and broadcasts a revision-transaction Tx'. Before including the revision-transaction in a block, a miner verifies if the current timestamp is less than the time-lock T mentioned in the previous transaction Tx (i.e., whether Tx is still unspendable).

We note that the time-locked transaction-rewriting mechanism is meaningful in practice. For example, in a credit card payment system, a transaction can be locked for a specific time period, during which the transaction amount may be modified (e.g., for adjusting a discount amount over an item sold) but the transaction modifier of the money (the payee in this case) cannot spend it. However, after the locking period is over, the transaction amount becomes spendable but it cannot be modified further. Similarly, for transactions made towards provident funds, the payer should be able to modify the amount of a transaction within a specified deadline (e.g., a deadline set by the government), but the payer is restricted to choose from the pre-defined set of messages only.

B. Mutable Blockchains Using MCH

Now we show how MCH can be used to rewrite blockchains both in the permissioned and permissionless settings. Here we provide a generic description that works with both the settings. In a blockchain, each block is a compact representation of a set of valid transactions stored at the leaf-nodes of a Merkle tree [20]. The hash of the root-node of the tree (say, TX_ROOT) corresponding to a block acts as a commitment to all the transactions included in that particular block. MCH enables one to rewrite some parts of the transactions without changing TX_ROOT. Consequently, the hash values of all the subsequent blocks present in the blockchain also remain intact.

We assume that each party in the network has a chameleon secret key $\mathtt{sk}$ and the corresponding public key is $\mathtt{pk}$. If a payer (transaction owner) wants to include a mutable content (or message) m along with other immutable content (say, $\bar{m}$) in a transaction, then the transaction includes $\mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m)$ along with the immutable content $\bar{m}$, where $\mathtt{pk}$ is the public key of the transaction modifier, and $\mathtt{ask}$ is the assertion secret key shared between the transaction owner and the transaction modifier (the corresponding assertion public key is $\mathtt{apk}$). Subsequently, all mutable transactions and normal transactions to be included in a block are hashed using the conventional collision resistant hash function $\mathtt{H}$. Figure 6 illustrates a block B_i containing four transactions $\mathsf{Tx}_{(i,1)}, \mathsf{Tx}_{(i,2)}, \mathsf{Tx}_{(i,3)}$, and $\mathsf{Tx}_{(i,4)}$, where $\mathsf{Tx}_{(i,1)}$ and $\mathsf{Tx}_{(i,3)}$ are mutable transactions chosen by the transaction owner, $\mathsf{Tx}_{(i,2)}$ and $\mathsf{Tx}_{(i,4)}$ are normal transactions. In particular, the mutable transactions $\mathsf{Tx}_{(i,1)}$ and $\mathsf{Tx}_{(i,3)}$ include the ("simplified" version of) chameleon

hash tuples $(h_{(i,1)}, r_{(i,1)}, c) = \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m_{(i,1)})$ and $(h_{(i,3)}, r_{(i,3)}, c) = \mathsf{Hash}(\mathtt{pk}, \mathtt{ask}, m_{(i,3)})$ along with the other immutable contents $\bar{m}_{(i,1)}$ and $\bar{m}_{(i,3)}$, respectively. The payer also includes a set of candidate messages M in a transaction (as auxiliary data). Depending on a specific application, the set M is signed by the payer (or the payee), if the payee (or the payer) acts as the transaction modifier.

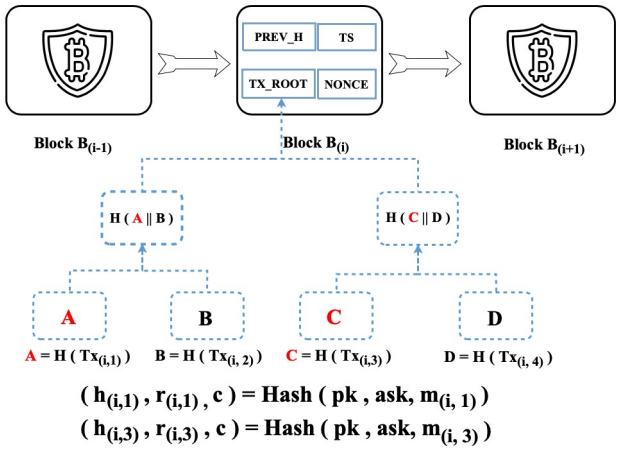

Fig. 6: The hash values A, B, C, and D are derived from collision resistant hash function $\mathtt{H}$ (e.g., $A = \mathtt{H}(\mathsf{Tx}_{(i,1)})$). The mutable transactions $\mathsf{Tx}_{(i,1)}$ and $\mathsf{Tx}_{(i,3)}$ include the chameleon hash tuples which are derived from Hash algorithm.

When a mutable transaction (more precisely, its mutable content or message) needs to be modified, the transaction modifier with the corresponding secret key $\mathtt{sk}$ can compute a collision in the chameleon hash $h_{(i,1)}$ (or $h_{(i,3)}$) for a new randomness value $r'_{(i,1)}$ (or $r'_{(i,3)}$). We note that the randomness values $r'_{(i,1)}$ and $r'_{(i,3)}$ are not included in the hash computations, and they are included as non-hashed parts of the transaction. As modifying a mutable content of a transaction does not change the corresponding hash due to the collision, TX_ROOT computed over all the transactions present in the corresponding block also remains the same as before. Moreover, $PREV\_H$ value present in the next block in the blockchain does not change, no further modifications are thus required in any of the subsequent blocks in the blockchain.

If certain contents in mutable transactions are hashed using MCH, then the following properties hold: 1) *indistinguishability* — given a transaction containing a mutable content, any outsider cannot distinguish whether the content is the original one or it is modified using MCH; 2) *collision-resistance* — a transaction modifier with rewriting permission can perform modifications only if the chosen new content belongs to set M selected and signed by the other party.

While rewriting a valid transaction Tx already included in the blockchain with a time-lock t, the transaction modifier chooses one message from the message set M Then, the transaction modifier generates a revision-transaction Tx' having the same structure as Tx, except that the new changes in transaction amounts are incorporated in Tx' (see Figure 7). We note that Tx' points to the previous transaction Tx (e.g., by using the transaction id of Tx), and the revision-transaction Tx' contains a special bit $b = 1$ that distinguishes it from a normal transaction (this special bit b is set to 0 in normal transactions). We also note that the main purpose of a revision-transaction is to help rewriting a previous block in the blockchain. In other words, one cannot spend a revision-transaction (with $b = 1$), i.e., any other transaction having a revision-transaction in its input script is not considered as valid.

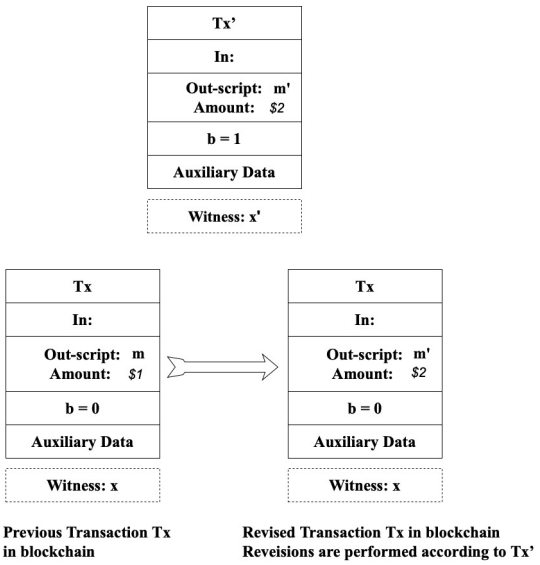

Fig. 7: For a transaction Tx with $b = 0$, miners treat it as a normal transaction to be included in a block in the blockchain. On the other hand, a revision-transaction Tx' indicates that a certain previous transaction Tx needs to be modified (due to changes in the data m' present in out-script or the transaction amount $\ddot{\mathbb{B}}2$). We note that a revision-transaction Tx' with $b = 1$ is always unspendable, and a normal transaction Tx with $b = 0$ is unspendable unless its time-lock has expired. A witness x denotes a digital signature computed on the transaction using the corresponding secret key [2].

Finally, the transaction modifier broadcasts Tx' to all the peer nodes in the blockchain. Tx' is included in the blockchain (by some miner) in the same way as a normal transaction is included (i.e., through the consensus). Before including the revision-transaction Tx' (with $b = 1$) in a potential block, a miner checks:

1) whether Tx' is a valid transaction pointing to another valid transaction Tx already present in the blockchain.
2) whether the current timestamp is less than the time-lock t mentioned in the previous transaction Tx (i.e., whether Tx is still unspendable);
3) whether $(\ddot{\mathbb{B}}1, \ddot{\mathbb{B}}2) \in \mathsf{M}$ (i.e, whether the old and new monetary values in Tx and Tx' are selected from the same message set M.
4) whether $1 = \mathsf{Verify}(\mathtt{pk}, \mathtt{apk}, \ddot{\mathbb{B}}2, h, r')$, where $\mathtt{pk}$ denotes

the transaction modifier's chameleon public key, apk denotes the assertion public key.

Once Tx' is included in the blockchain through consensus, every node in the network modifies the previous transaction Tx (more precisely, the mutable monetary value in it) according to the changes specified in Tx'. After time t, the updated transaction Tx becomes spendable — which eliminates the possibility of transaction inconsistency. We emphasize that we rely mostly on the existing consensus mechanism for registering transactions (both normal transactions and revision-transactions) into the blockchain — which requires small changes in order to incorporate rewriting.

Comparison with [14]. In [14], Deuber et al. proposed a consensus-based e-voting system to integrate redactable transactions into the blockchain. Informally, a user first broadcasts a modification request to the network, which includes a candidate block. Then, there is a voting period when all miners may vote for the candidate block. Eventually, if the request is approved (i.e., a certain number of votes are received in its favor), then the original block will be replaced by the candidate block. We summarize the advantages of our techniques over those of [14] as follows: 1) Rewriting the blockchain in our case requires small changes in the existing consensus mechanism (e.g., transactions include fields b and auxiliary data, and the validation procedure for mutable transactions includes MCH verification as well). However, we do not need any separate mechanism to incorporate changes in the blockchain. On the contrary, [14] relies on a separate consensus-based e-voting mechanism to handle modifications in the blockchain (along with the existing consensus for selecting new blocks to be appended to the blockchain). 2) The permissible changes both in our scheme and [14] are fine-grained as the content of a transaction (except its transaction amount) can be modified. However, changes in the monetary amounts of a transaction are allowed in our case but *not* in [14].

Limitations. Our techniques for rewriting a blockchain have the following limitations. 1) The blockchain records the revision histories of all mutable transactions. 2) In order to maintain a synchronized view of the blockchain with respect to its peers, each node storing a copy of the blockchain should modify original mutable transactions according to revision-transactions. While it is difficult to enforce that each node in the blockchain updates its transactions, the whole system works if a majority of nodes follow its protocol as in the original blockchain system. Nonetheless, a malicious node can always keep separate copies of the original transactions. However, we note that all existing schemes, that rewrite blockchains, suffer from this limitation. 3) Our solution focuses on systems based on unspent transaction outputs or UTXO [8] (e.g., Bitcoin and some of its derivatives such as Zcash [25], Litecoin [5]), where unspent transactions are stored in a list and they are spent in an all-or-nothing fashion. We use the time-lock provision available in Bitcoin transactions in order to handle transaction inconsistency (for changes in monetary values). The time-lock for a mutable transaction in our scheme denotes a time-bound that the message of the transaction (e.g., monetary values) must be finalized within. We note that this restriction is not suitable for certain applications where changes need to be made irrespective of such time-bounds.

VII. Conclusion

In this paper, we proposed the first generic framework for blockchain rewriting based on message-controlled chameleon hash MCH and applied it to address the issue of arbitrary or malicious rewriting in blockchains. Our framework enforces that a permitted party should be able to replace a message already in a blockchain by a message chosen from a specified set of messages only. Unlike the previous blockchain rewritings, which support blockchain rewriting at either block-level or transaction-level, in either permissioned setting or permissionless setting, our framework allows blockchain rewriting at message-level in both settings.

References

[1] Bitcoin Developer Guide. https://bitcoin.org/en/developer-guide.
[2] Bitcoin Script. https://en.bitcoin.it/wiki/Script.
[3] Block Timestamp. https://en.bitcoin.it/w/index.php? title= Block_timestamp&oldid=51392.
[4] General Data Protection Regulation. https://gdpr-info.eu.
[5] Litecoin. https://litecoin.org.
[6] Rewritable Chain. https://github.com/SMC-SMU/rewritableBlockchain.
[7] Trapdoor Function. https://en.wikipedia.org/wiki/Trapdoor_function.
[8] UTXO. https://en.wikipedia.org/wiki/Unspent_transaction_output.
[9] G. Ateniese and B. de Medeiros. On the key exposure problem in chameleon hashes. In *SCN*, pages 165–179. Springer, 2004.
[10] G. Ateniese, B. Magri, D. Venturi, and E. Andrade. Redactable blockchain–or–rewriting history in bitcoin and friends. In *EuroS&P*, pages 111–126, 2017.
[11] J. Camenisch, D. Derler, S. Krenn, H. C. Pöhls, K. Samelin, and D. Slamanig. Chameleon-hashes with ephemeral trapdoors. In *PKC*, pages 152–182, 2017.
[12] D. Derler, S. Ramacher, and D. Slamanig. Short double-and n-times-authentication-preventing signatures from ecdsa and more. In *EuroS&P*, pages 273–287, 2018.
[13] D. Derler, K. Samelin, D. Slamanig, and C. Striecks. Fine-grained and controlled rewriting in blockchains: Chameleon-hashing gone attribute-based. In *NDSS*, 2019.
[14] D. Deuber, B. Magri, and S. A. K. Thyagarajan. Redactable blockchain in the permissionless setting. In *IEEE S&P*, pages 124–138, 2019.
[15] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.
[16] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *FOCS*, pages 427–438, 1987.
[17] J. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310, 2015.
[18] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *IEEE S&P*, pages 27–40, 2004.
[19] H. Krawczyk and T. Rabin. Chameleon signatures. In *NDSS*, 2000.
[20] R. C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.
[21] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.
[22] B. Poettering and D. Stebila. Double-authentication-preventing signatures. In *ESORICS*, pages 436–453, 2014.
[23] I. Puddu, A. Dmitrienko, and S. Capkun. μchain: How to forget without hard forks. *IACR Cryptology ePrint Archive*, 2017:106, 2017.
[24] T. Ruffing, A. Kate, and D. Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of bitcoins. In *CCS*, pages 219–230, 2015.
[25] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE S&P*, pages 459–474, 2014.

[26] D. Schröder and H. Schröder. Verifiable data streaming. In *CCS*, pages 953–964, 2012.
[27] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
[28] S. A. K. Thyagarajan, A. Bhat, B. Magri, D. Tschudi, and A. Kate. Reparo: Publicly verifiable layer to repair blockchains. *arXiv preprint arXiv:2001.00486*, 2020.
[29] Y. Tian, N. Li, Y. Li, P. Szalachowski, and J. Zhou. Policy-based chameleon hash for blockchain rewriting with black-box accountability. In *ACSAC*, pages 813–828, 2020.

APPENDIX A
SECURITY OF MCH

The security analysis of MCH includes indistinguishability, and collision-resistance.

A. Indistinguishability

Theorem 6: MCH achieves indistinguishability if eAA and semi-public-coin CH are indistinguishable.

Proof 1: We define a sequence of games $\mathbb{G}_i$, $i = 0, \cdots, 2$ and let $\mathtt{Adv}_i$ denote the advantage of the adversary in game $\mathbb{G}_i$. Assume that $\mathcal{A}$ activates at most $n(\lambda)$ chameleon hash queries in each game, and let $g \in [1, n(\lambda)]$ be a guess for the index of the HashOrAdapt query.

- $\mathbb{G}_0$: This is original game for indistinguishability.
- $\mathbb{G}_1$: This game is identical to game $\mathbb{G}_0$ except that in the g-th query, $\mathcal{S}$ directly hashes a challenge message in the construction of an assertion, instead of calculating the chameleon hash using assertion secret keys. Below we show that the difference between $\mathbb{G}_0$ and $\mathbb{G}_1$ is negligible if eAA is indistinguishable.
  Let $\mathcal{S}$ be a distinguisher against eAA, who is given an assertion public key $\mathtt{apk}^*$ and a HashOrAdapt oracle, aims to break the indistinguishability of eAA. $\mathcal{S}$ sets the assertion public key in the g-th query as $\mathtt{apk}^*$, and generates the chameleon key pair honestly.
  If $\mathcal{A}$ submits two messages $(m_0, m_1) \in \mathsf{M}$ in the g-th query, $\mathcal{S}$ invokes the HashOrAdapt oracle k times to obtain an assertion τ_b on message m_b. Next, $\mathcal{S}$ returns the chameleon hash (h_b, r_b, c) to $\mathcal{A}$, where $r_b = (\xi_b, C_\rho, \tau_b, \mathtt{pk}')$, $(h_b, \xi_b) \leftarrow \mathsf{CH.Hash}(\mathtt{pk}', m_b; \rho)$, $C \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho)$. Note that $\mathtt{pk}'$ is derived from $\mathtt{pk}_c$ and c. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. If $\mathcal{A}$ guesses the random bit correctly, then $\mathcal{S}$ can break the indistinguishability of eAA. Since at most k adapted randomness involved in an assertion, hence we have
$$|\mathtt{Adv}_0 - \mathtt{Adv}_1| \leq k \cdot \mathtt{Adv}_{\mathcal{S}}^{\mathrm{eAA}}(\lambda). \quad (1)$$
- $\mathbb{G}_2$: This game is identical to game $\mathbb{G}_1$ except that in the g-th query, $\mathcal{S}$ directly hashes a challenge message, instead of calculating the chameleon hash using a chameleon secret key. Below we show that the difference between $\mathbb{G}_1$ and $\mathbb{G}_2$ is negligible if the semi-public-coin CH scheme is strongly indistinguishable.
  Let $\mathcal{S}$ be a distinguisher against CH, who is given a chameleon secret key $\mathtt{sk}^*$ and a HashOrAdapt oracle, aims to break the strong indistinguishability of CH [13], [29]. $\mathcal{S}$ sets up the CH secret key as $\mathtt{sk}^*$, and simulates the remaining parameters honestly.
  If $\mathcal{A}$ submits two messages $(m_0, m_1) \in \mathsf{M}$ in the g-th query, $\mathcal{S}$ first obtains a chameleon hash (h_b, ξ_b) from his HashOrAdapt oracle on message m_b. Then, $\mathcal{S}$ simulates the assertion $\tau_b \leftarrow \mathsf{eAA.Assert}(\mathtt{ask}', \mathtt{auxsk}', m_b)$, where $(\mathtt{ask}', \mathtt{auxsk}', \mathtt{apk}) \leftarrow \mathsf{eAA.KeyGen}(1^\lambda, \mathsf{M})$. Eventually, $\mathcal{S}$ returns (h_b, r_b, c) to $\mathcal{A}$, where $r_b = (\xi_b, C_b, \tau_b, \mathtt{pk}')$, $C_b \leftarrow \mathsf{PKE.Enc}(\mathtt{pk}_e, \rho_b)$. Note that $\mathtt{pk}'$ is derived from $\mathtt{pk}^*$ and c. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. If $\mathcal{A}$ guesses the random bit correctly, then $\mathcal{S}$ can break the indistinguishability of CH. Therefore, we have
$$|\mathtt{Adv}_1 - \mathtt{Adv}_2| \leq \mathtt{Adv}_{\mathcal{S}}^{\mathrm{CH}}(\lambda). \quad (2)$$

Combining the above results together, we have
$$\mathtt{Adv}_{\mathcal{A}}(\lambda) \leq n(\lambda)(k \cdot \mathtt{Adv}_{\mathcal{S}}^{\mathrm{eAA}}(\lambda) + \mathtt{Adv}_{\mathcal{S}}^{\mathrm{CH}}(\lambda)).$$

B. Collision Resistance

Theorem 7: MCH achieves collision-resistance if eAA and CH are collision resistant.

Proof 2: We define a sequence of games $\mathbb{G}_i$, $i = 0, \cdots, 3$, and let $\mathtt{Adv}_i$ denote the advantage of the adversary in game $\mathbb{G}_i$. We assume that $\mathcal{A}$ activates at most $n(\lambda)$ number of queries to Hash oracle in each game.

- $\mathbb{G}_0$: This is original game for collision-resistance.
- $\mathbb{G}_1$: This game is identical to game $\mathbb{G}_0$ except the following difference: $\mathcal{S}$ randomly chooses $g \in [1, n(\lambda)]$ as a guess for the index of the forgery such that $\mathcal{A}$ outputs $(h^*, m^*, r^*, m'^*, r'^*)$ under a chameleon public key $\mathtt{pk}^*$ and an assertion public key $\mathtt{apk}^*$. $\mathcal{S}$ will output a random bit if $\mathcal{A}$'s forgery does not occur in the g-th query. Therefore we have
$$\mathtt{Adv}_1 = n(\lambda) \cdot \mathtt{Adv}_0 \quad (3)$$
- $\mathbb{G}_2$: This game is identical to game $\mathbb{G}_1$ except that $\mathcal{S}$ will output a random bit if $\mathcal{A}$ outputs a valid forgery in eAA. Below we show that the difference between $\mathbb{G}_1$ and $\mathbb{G}_2$ is negligible if eAA is collision resistant.
  Let $\mathcal{S}$ be an attacker against eAA, who is given an assertion public key $\mathtt{apk}^*$ and an Assert' oracle, aims to find a forgery. $\mathcal{S}$ simulates the game for $\mathcal{A}$ as follows.
  - $\mathcal{S}$ sets up the assertion public key in the g-th query as $\mathtt{apk}^*$. For the non g-th queries, $\mathcal{S}$ generates a new assertion key pair for an assertion and proceeds honestly, storing the corresponding assertion key pairs.
  - For the g-th query, $\mathcal{S}$ sets $\mathtt{apk}^*$ as the assertion public key, invokes his Assert' oracle to obtain τ, and outputs the chameleon hash tuple as (h, m, r, c), where $r = (\xi, C_\rho, \tau, \mathtt{pk}')$. In particular, the chameleon hash (h, ξ) can be simulated by $\mathcal{S}$.
  - If $\mathcal{A}$ outputs a chameleon hash tuple $(h^*, m^*, r^*, m'^*, r'^*)$ such that the assertion τ'^* is valid on message m'^* under assertion public key $\mathtt{apk}^*$, and it was not previously returned by the Assert' oracle, then $\mathcal{S}$ outputs $(m^*, \tau^*, m'^*, \tau'^*)$ as the forgery attempt to eAA. Meanwhile, the chameleon hash $(h^*, m^*, r^*, m'^*, r'^*)$ and the

assertion τ^* can be previously simulated by $\mathcal{S}$. Therefore, we have

$$|\mathtt{Adv}_1 - \mathtt{Adv}_2| \leq \mathtt{Adv}_{\mathcal{S}}^{\mathrm{eAA}}(\lambda). \tag{4}$$

- $\mathbb{G}_3$: This game is identical to game $\mathbb{G}_2$ except that in the g-th query, $\mathcal{S}$ will output a random bit if $\mathcal{A}$ outputs a forgery in CH. Below we show that the difference between $\mathbb{G}_2$ and $\mathbb{G}_3$ is negligible if CH is collision resistant.
  Let $\mathcal{S}$ be an attacker against CH, who is given a chameleon public key $\mathtt{pk}^*$, a Hash oracle and an Adapt' oracle, aims to find a forgery. $\mathcal{S}$ includes the valid messages into a set $\mathcal{Q}$ when invoking Hash oracle. $\mathcal{S}$ simulates the game for $\mathcal{A}$ as follows.
  - $\mathcal{S}$ sets up the chameleon public key in the g-th query as $\mathtt{pk}^*$. For non g-th queries, $\mathcal{S}$ generates a new chameleon key pair for a chameleon hash and proceeds honestly, storing the corresponding chameleon key pairs.
  - For the g-th query, $\mathcal{S}$ sets $\mathtt{pk}^*$ as the chameleon public key, invokes his Hash oracle to obtain (h, ξ) and outputs the chameleon hash tuple as (h, m, r, c), where the assertion τ can be simulated by $\mathcal{S}$ according to the protocol specification. In particular, $\mathcal{S}$ includes $(\mathtt{pk}', h, m, r, c)$ into $\mathcal{Q}$. Note that $\mathtt{pk}'$ is derived from $\mathtt{pk}^*$ and c.
    If $\mathcal{A}$ issues an adapt query in the form of $(\mathtt{pk}', c, h, m, r, m')$, $\mathcal{S}$ checks whether $(\mathtt{pk}', h, m, r, c)$ was previously simulated ($\mathcal{S}$ aborts if not). Then, $\mathcal{S}$ invokes his Adapt' oracle to obtain r' if $m, m' \in \mathsf{M}$ and c is established between (m, m'), and returns the chameleon hash $(\mathtt{pk}, h, m', r')$ to $\mathcal{A}$. In particular, the assertion τ' can be simulated by $\mathcal{S}$.
  - If $\mathcal{A}$ outputs a chameleon hash tuple that includes a chameleon hash (h^*, r'^*), which is valid on message m'^* under the chameleon public key $\mathtt{pk}'$ and the assertion public key $\mathtt{apk}$, and it was not previously returned by the Adapt' oracle. In particular, the condition is satisfied $(m^*, m^{'*}) \in \mathsf{M}$, and the relationship between the two messages is established. Then, $\mathcal{S}$ outputs $(\mathtt{pk}', m^*, r^*, m'^*, r'^*)$ as the forgery attempt to CH. Therefore, we have

$$|\mathtt{Adv}_2 - \mathtt{Adv}_3| \leq \mathtt{Adv}_{\mathcal{S}}^{\mathrm{CH}}(\lambda). \tag{5}$$

Combining the above results together, we have

$$\mathtt{Adv}_{\mathcal{A}}(\lambda) \quad \leq \quad n(\lambda)(\mathtt{Adv}_{\mathcal{S}}^{\mathrm{eAA}}(\lambda) + \mathtt{Adv}_{\mathcal{S}}^{\mathrm{CH}}(\lambda)).$$